1997

Protocols for secure client-server applications in the Joint Maritime command Information System

Weldon, Steven G

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/8869

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PROTOCOLS FOR SECURE CLIENT-SERVER APPLICATIONS IN THE JOINT MARITIME COMMAND INFORMATION SYSTEM

by

Steven G. Weldon

September, 1997

Thesis Advisor:	Dennis Volpano
Second Reader:	Cynthia Irvine

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 1997	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
PROTOCOLS FOR SECURE CLIENT-SERVER APPLICATIONS IN THE JOINT MARITIME COMMAND INFORMATION SYSTEM.

5. FUNDING NUMBERS

6. AUTHOR(S)
Weldon, Steven G.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

ABSTRACT *(maximum 200 words)*

The new architecture for the Joint Maritime Command Information System, referred to as JMCIS'98, seeks to provide uniform access to tactical and non-tactical information. The goal is to allow access to such information using Wide Area Network technology and Personal Computers running Windows NT in a web environment. This architecture relies on web servers to deliver executable content, such as Java applets, to clients and gateway servers to route requests to the appropriate servers and/or databases.

This architecture raises new security risks which must be addressed. Two of these risks are addressed in this thesis: executing downloaded code from a web server and transmitting sensitive information, such as passwords, to gateway servers

We investigate three cryptographic protocols to address these risks. The first protocol treats the risk of executing downloaded code from a web server by using digital signatures. The second protocol addresses the transmission of sensitive information to a gateway server by using certificates and symmetric key cryptography. Finally, we explore an alternative approach, that of the Secure Sockets Layer, which provides mutual authentication. We discuss how the first two protocols can be implemented in Java using the Java Developer's Kit (JDK) 1.1 and the Java Cryptography Extension (JCE) 1.1.

14. SUBJECT TERMS
Protocols, JMCIS'98, Secure Client-Server Applications

15. NUMBER OF PAGES
71

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

PROTOCOLS FOR SECURE CLIENT-SERVER APPLICATIONS IN THE JOINT MARITIME COMMAND INFORMATION SYSTEM

Steven G. Weldon
Lieutenant, United States Navy
B.A., The George Washington University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

ABSTRACT

The new architecture for the Joint Maritime Command Information System, referred to as JMCIS'98, seeks to provide uniform access to tactical and non-tactical information. The goal is to allow access to such information using Wide Area Network technology and Personal Computers running Windows NT in a web environment. This architecture relies on web servers to deliver executable content, such as Java applets, to clients and gateway servers to route requests to the appropriate servers and/or databases.

This architecture raises new security risks which must be addressed. Two of these risks are addressed in this thesis: executing downloaded code from a web server and transmitting sensitive information, such as passwords, to gateway servers.

We investigate three cryptographic protocols to address these risks. The first protocol treats the risk of executing downloaded code from a web server by using digital signatures. The second protocol addresses the transmission of sensitive information to a gateway server by using certificates and symmetric key cryptography. Finally, we explore an alternative approach, that of the Secure Sockets Layer, which provides mutual authentication. We discuss how the first two protocols can be implemented in Java using the Java Developer's Kit (JDK) 1.1 and the Java Cryptography Extension (JCE) 1.1.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This research was possible due to the efforts of many people. Most notably is that of my thesis advisors. The individual attention, patience, professionalism and enthusiasm displayed by my thesis advisors, Professors Dennis Volpano and Cynthia Irvine, allowed me to accomplish things that seemed so far away, yet were closer than I thought.

Sincere thanks go out to several of my fellow students and friends. Cpt. Ken Fritzsche, USA and Cpt. Jeff May, USA for your positive guidance and willingness to help a shipmate. LT Marcia Edmiston, USN for your support and advice. LT Bart Umentum, USN for a sympathetic ear on all matters from thesis work to baseball. I consider you all the finest of friends.

I would also like to thank the students and faculty members at the Naval Postgraduate School, I have found the educational experience to be a positive one and this is only possible with the outstanding faculty and peers with which this institution is blessed.

Finally, I am forever grateful to my mother, Melba Weldon, for her love, patience, understanding and support throughout this research and my studies.

I. INTRODUCTION

The Joint Maritime Command Information System (JMCIS) is the United States Navy's primary command and control system. The components of JMCIS allow ashore, afloat and tactical/mobile users to communicate. Currently, the JMCIS architecture is designed around UNIX servers and clients. The next generation of the JMCIS, named JMCIS'98, seeks to use many of the recent revolutionary changes in the network computing environment. Its aim is to provide a single, consistent architecture which will cross sea and shore, as well as tactical and non-tactical boundaries. The JMCIS'98 architecture is being built around commercial-off-the-shelf (COTS) technology, specifically Microsoft Windows NT clients and servers operating in a web environment.

The JMCIS'98 Network Architecture is composed of clients, web servers, gateway servers, access servers and databases. A typical scenario is one in which a client connects to a web server and downloads executable content to accomplish a task. This executable content may need to connect the client to a gateway server in order to query a tactical database. The client sends database queries to the gateway server which then forwards them to an appropriate access server. The access server responds to queries by accessing the database. Query replies are then sent back to the client via the gateway server.

The Java programming language has been used extensively in the re-engineering of the JMCIS system. Java programs can be written as either applications or applets. Java applets are intended to be embedded in a web page residing on a web server, downloaded by a client across the network and then executed on the client's machine. The functionality of JMICIS'98 will, to a great extent, be centered around Java applets. Much of a client's interaction with JMCIS'98 will be via these Java applets. By using applets downloaded from a web server for client functionality, JMCIS'98 will have a 'fat server, thin client' architecture. This fat server, thin client architecture is expected to dramatically ease system administrator efforts in such areas as configuration management and maintenance. This approach also has the added benefit of being dynamically configurable, by allowing a single change to the code on the server to affect all users, thereby making time-consuming and expensive changes on each individual machine unnecessary.

A. SECURITY ISSUES IN JMCIS'98

There is a pressing need for the Navy to provide security for its widely distributed networks. Commanders must improve their ability to secure their information infrastructure while simultaneously expanding its use in a global environment. Security is an important consideration in the JMCIS'98 evolution. This thesis brings to light a few of the relevant security considerations and some ways they can be addressed in today's operating environment.

The JMCIS'98 architecture is built around a client downloading executable content from a web server. Thus, as part of a safe, secure computing environment there needs to be a means by which the client can establish trust in the code based on the origin of the code. By being able to trust the code, the client can execute it with reasonable assurance that the code is not malicious.

There will also be situations when the client and server will need to securely exchange sensitive information. The client may need to connect to a gateway server for the purpose of sending sensitive data, such as the client's name and password, as part of a tactical database query. The gateway server will send this information to the appropriate access server. The access server will check that the client is an authorized JMCIS'98 user and will then satisfy the request by passing the query to a database. The query results are then forwarded back to the client, via the access and gateway servers. Sensitive information, such as the client's name, password, the query and the query response, should be protected during transmission across the network.

The goal of this thesis is to introduce two protocols that are intended for use as part of the JMCIS'98 architecture and to discuss their implementation using the Java programming language. These protocols, developed by Dennis Volpano at the Naval Postgraduate School, are intended to allow for safe, reliable transfer of information between clients and servers in a widely distributed networking environment.

Specifically, this work will introduce a Trusted Code Protocol designed to allow a client to download trusted executable code from a server and safely execute this code. An implementation of the Trusted Code Protocol using Java's Cryptography Architecture is given. An alternative approach to verifying trusted code using the Java Developer's Kit 1.1 functionality is also discussed.

A Secure Password Transmission Protocol is also introduced. This protocol is designed to allow a client to make secure information exchanges with a server.

Information that may be passed in this situation includes such sensitive items as passwords for establishing client authentication, thus confidentiality of this information is required. An implementation of the this protocol, using Java's Cryptography Extension, is developed in this thesis. An alternative to secure password transmission using Netscape's Secure Sockets Layer technology is also discussed.

B. ORGANIZATION OF THESIS

Chapter II of this thesis describes the Joint Maritime Command Information System, including its architecture and the changes that comprise the JMCIS'98 effort. Chapter III provides a general overview of the Java programming language and its use in applications and applets. Chapter IV provides a description of the cryptographic support included in the Java Developers Kit version 1.1, the Java Cryptography Architecture and the Java Cryptography Extension version 1.1. Chapter V introduces the Trusted Code and Secure Password Transmission Protocols and sketches their implementations using the Java Cryptography Architecture and the Java Cryptography Extension. An alternative to verifying trusted code, using the Java Developer's Kit 1.1, and an alternative to secure password transmission using Netscape's Secure Sockets Layer are also discussed in Chapter V. Chapter VI presents conclusions and recommendations for further work.

II. JMCIS'98

The Joint Military Command Information System (JMCIS) is the United States Navy's primary Command and Control system. JMCIS is undergoing an evolutionary and revolutionary change to rapidly develop and field new Command, Control, Communications, Computers and Intelligence (C4I) capabilities for Navy Afloat, Ashore and Tactical/Mobile users. JMCIS'98 is the term used to describe the current evolutionary efforts to the JMCIS architecture. The driving forces behind the JMCIS'98 strategy are:

- The migration of JMCIS to the Defense Information Infrastructure (DII).
- Fleet requirements for merging tactical and non-tactical networks.
- Emerging Web and Personal Computer (PC) technologies to provide required information/capabilities.

Migration to the DII Common Operating Environment (COE) is expected to bring true interoperability across all the services. With the move to DII compliance, JMCIS will be the naval implementation of the Global Command and Control System (GCCS).

The merging of Tactical and Non-Tactical (TnT) networks enables the fleet users to perform both tactical and non-tactical tasks on a single workstation. This is intended to eliminate redundancies and simplify training and system administration tasks.

Emerging Web and PC technologies allow for the migration of the JMCIS architecture to PC servers and clients. JMCIS'98 planning calls for an incremental shift from UNIX-based servers and clients to Personal Computer-based servers and clients. This move is expected to yield numerous life-cycle cost benefits such as reducing the fielding time of new computer systems, reduced training costs and is also in line with other Department of Defense and Navy focus areas such as the extended use of Commercial-Off-The-Shelf (COTS) products and Web technology.

JMCIS'98 represents a significant departure from "business as usual" through which virtually all program processes, including acquisition, are being integrated and streamlined. The JMCIS'98 effort includes a common strategy and implementation plan for all components of JMCIS:

- Navy Tactical command System-Afloat (NTCS-A) is the afloat component;
- Operations Support System (OSS) is the ashore component, and;

- Tactical Support Center And Tactical Mobile Variants (TSC/TMV) is the tactical/mobile component.

The JMCIS program office (PMW-171) has been tasked, whenever practical, to implement functionality using commercial standards and products to reduce cost and increase productivity of JMCIS development, testing and training. This strategy will enable JMCIS to satisfy Fleet requirements faster and more effectively, and is expected to reduce the current backlog of requirements.

In addition to migrating to the DII COE, JMCIS'98 builds will be fielded in conjunction with equipment installations of the Information Technology for the 21st Century (IT-21) initiative whenever available resources exist for IT-21. Phase 1 of IT-21 provides an initial LAN and communications capability. Phase 2 adds additional equipment and communications infrastructure to the capability including a full Asynchronous Transfer Mode (ATM) backbone and large numbers of Windows NT clients and Windows NT servers. [Ref. 1]

A. JMCIS'98 ARCHITECTURE

The JMCIS'98 network architecture centers around a client which accesses web, gateway and access servers and their associated database(s). The functionality of this architecture describes how a client can make a connection to the network through the web server and download executable content. The client would then use this downloaded code to make connections to a gateway server. The gateway server's function is to route the client's database query to the appropriate access server. The access server will authenticate the client. Assuming that the client's authentication is carried out successfully, the access server will route the query to the appropriate databases. The response to the query will then be forwarded back to the client via the access and gateway servers. Figure 2-1 illustrates the JMCIS'98 network architecture.

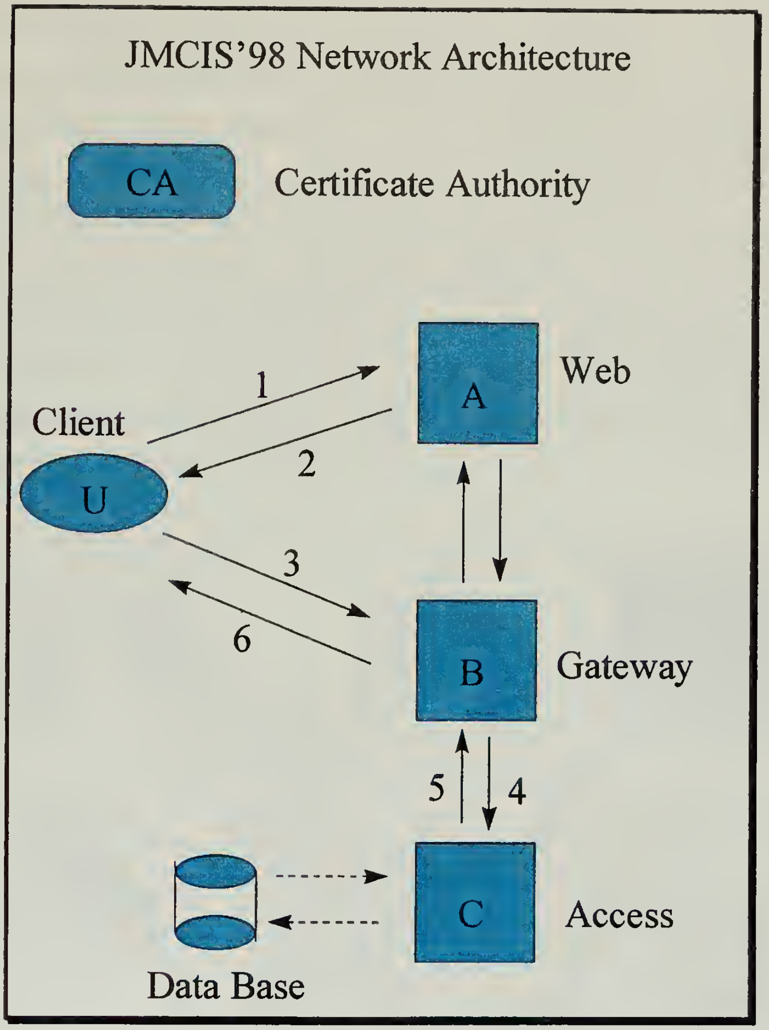


Figure 2-1. JMCIS'98 Network Protocol

Steps one and two of Figure 2-1 illustrate a client accessing the web server and downloading executable content. Steps three and four illustrate a client request being sent to the gateway server and routed to the access server for client authentication. The authenticated request is then passed to the appropriate database. Steps five and six illustrate the query results being passed from access server to the gateway server and finally, to the requesting client.

1. Migration Strategy

As part of the JMCIS'98 migration strategy, PMW-171 is investigating the use of commercial products which serve as a bridge between current and desired technology.

These products are intended to increase the initiative's flexibility in moving current UNIX applications to the PC and Web environments by enabling users with UNIX workstations to run PC applications, and vice versa. While products designed to bridge such gaps perform their intended function, they often accomplish this through a trade-off (e.g., high demand for local network bandwidth and additional processing time) which makes them undesirable for use within the target architecture. As such, they are viewed as temporary solutions which ensure continuity of operations during the transition period. Most will not be considered for permanent use within JMCIS'98. Figure 2-2 illustrates the JMCIS'98 architecture and depicts the manner in which Personal Computers will migrate into the traditional JMCIS architecture.

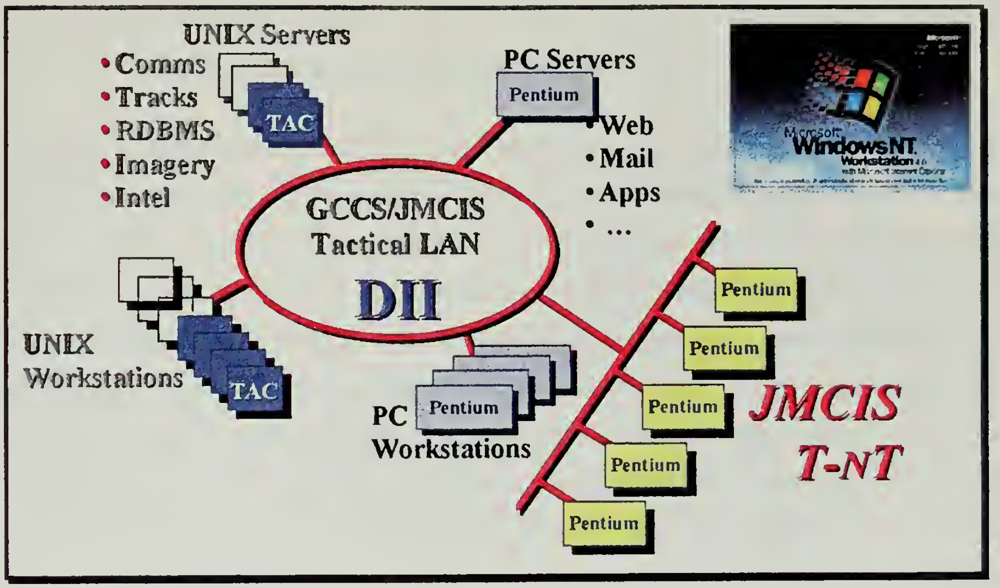


Figure 2-2. JMCIS'98 Architecture. [From Ref.1]

2. Personal Computer Architecture

Historically, the processing demands of JMCIS have made the power of UNIX and high-performance workstations and servers a necessity. Even though the prices of high-performance UNIX workstations have dropped slightly, they far exceed the cost of Personal Computers. At the same time, PCs have become quite powerful, approaching the performance of the UNIX workstations. The transition from UNIX workstations to PCs results in a number of significant, cost-related benefits:

- Lower cost per workstation will allow JMCIS to be implemented consistently on all ships, shore sites and JMCIS Tactical/Mobile sites;
- Lower cost per workstation will facilitate future periodic hardware replacement, vice forcing maintenance of equipment that is no longer in use in the commercial marketplace;
- Overall software costs for a PC-based JMCIS architecture are expected to be considerably less than the cost of maintaining large development, training and maintenance infrastructures for Government-Off-The-Shelf (GOTS) products;
- PCs offer an enormous selection of relatively inexpensive and user-friendly software applications, including development tools, and market pressure has resulted in a relatively uniform set of standards for sharing data;
- Many Fleet users are already accustomed to working on PCs, through school and home use, which reduces training costs. [Ref. 1]

In contrast, current JMCIS UNIX workstations are expensive and many users consider them difficult to use. Commercial hardware and software selection is limited and expensive. UNIX requires skilled system administrators and costly programming services to develop and maintain system software. Lack of user familiarity with UNIX and the complexity of UNIX training has led to performance and utilization shortfalls.

The migration to a PC architecture will make many tasks easier, and faster than previously possible. JMCIS software developers will be able to use a wide selection of commercial products and technologies, readily available in the PC market, to allow them to focus on writing JMCIS value added software code. They will spend less time and effort building infrastructure functionality and will be able to concentrate on putting the highest quality C4I products into the field. Examples of commercially-accepted technologies include the Common Object Model (COM), Object Linking and Embedding (OLE), the Distributed Computing Environment (DCE), Common Object Request Broker Architecture and graphical user interface (GUI) builders. By using these commercially accepted technologies, JMCIS'98 will easily integrate COTS software with code written specifically for JMCIS by JMCIS developers. With development taking advantage of re-use and extension of software code written and tested by commercial developers, the potential for creating errors is reduced drastically. This efficiency extends into testing, evaluation, training and technical support.

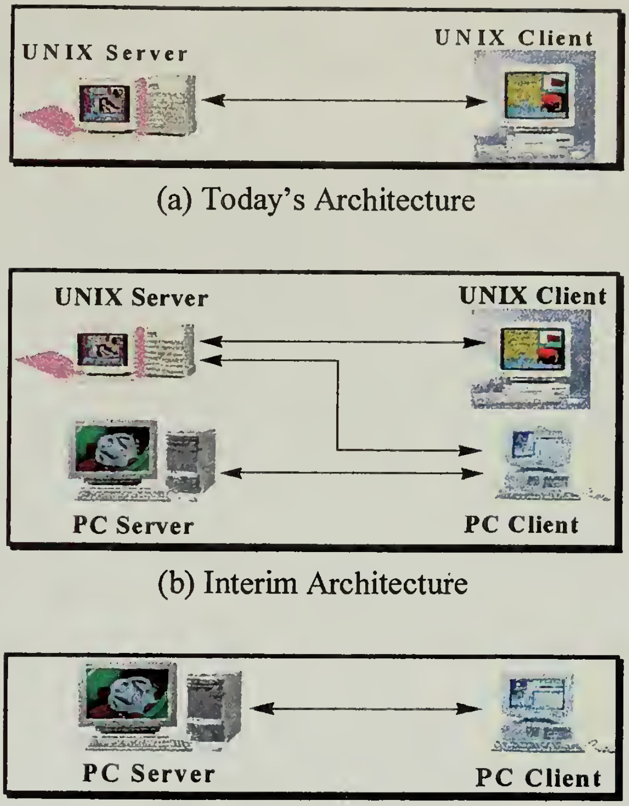
3. Communications

JMCIS presently provides C4I services to the Fleet both Afloat and Ashore. Afloat configurations can be categorized as force-level and unit-level configurations. Additionally, the Mobile Integrated Command Facility (MICFAC) is a mobile command facility designed to provide the Joint Task Force Commander with similar C4I capabilities when forward-deployed ashore in a theater of operations. Ashore configurations of JMCIS are located in Fleet command centers and JMCIS Tactical/Mobile. Future plans will include Type Commands (TYCOM) also. In order to allow for maximum interoperability among JMCIS and GCCS systems at all locations, Afloat and Ashore, JMCIS will adopt a single communications medium to the maximum extent possible. JMCIS will use the Secret IP Router Network (SIPRNET) and the Joint Worldwide Intelligence Communication System (JWICS) to provide the necessary wide-area network (WAN) connectivity. The Joint Maritime Communications System (JMCOMS) will provide the WAN connectivity for the Afloat and Tactical JMCIS systems. Operating "system–high" at the Secret and SCI security levels, SIPRNET and JWICS utilize the same protocols as the Internet (TCP/IP).

JMCIS locations will vary in how they connect to these networks. Command centers will connect through land-based high-bandwidth connections; battle-groups through satellite communication (SATCOM) and other circuits; and mobile facilities through a combination of SATCOM and dial-up (STU-IIIs, INMARSAT, etc.) provided by JMCOMS via the Advanced Digital Network System (ADNS).

4. Phased Migration

The JMCIS architecture onboard a ship or within a command will look much like it does today. One of the most significant differences, however, will be the phased replacement of UNIX servers with Windows NT servers and UNIX client workstations with Windows NT clients. In order to avoid using both a JMCIS PC and UNIX workstation to perform separate functions, the architecture will require a phased migration whereby segments are migrated in discrete groups (JUMPs). Figure 2-3 depicts the incremental stages by which today's JMCIS architecture will evolve into the planned JMCIS'98 architecture utilizing Windows NT clients and servers.



(a) Today's Architecture

(b) Interim Architecture

(c) Target Architecture

Figure 2-3. JMCIS Phased Migration to Personal Computers [From Ref. 1]

5. JMCIS Security

The architecture of JMCIS'98 poses new challenges regarding how the USN/USMC can continue to protect information from unauthorized access and compromise. JMCIS is increasing its reliance on WANs such as SIPRNET and JWICS. While the use of these networks unquestionably unlocks significant potential for interoperability and communications at all levels of command, it also increases the need to safeguard data. Integrity, Authentication, Trust, Denial of service attacks, packet "spoofing," password cracking, are examples of common security issues that must be addressed in order that Commanders may safely and efficiently use the JMCIS'98 architecture. In migrating to the PC environment, JMCIS recognizes the challenges posed by computer viruses and Trojan Horses, which are a significantly greater concern here than in the UNIX environment. The integration of tactical and non-tactical systems

will bring about additional security challenges. For example, non-tactical information systems, which typically operate at confidential or unclassified levels, contain data that is relevant to JMCIS.

The JMCIS'98 approach to information security is not to view it as a discrete process or life-cycle phase implemented as a development stage. Rather, information security will be practiced within all traditional life-cycle phases. It is being designed into the system and is an integral part of testing as well as practiced and inspected in operational JMCIS systems within the Fleet. The JMCIS'98 technical approach will increase the roles and responsibilities of security personnel. In order to thoroughly address the security issues inherent in the development of information systems, JMCIS security engineers are tasked with:

- Improving existing security architecture on UNIX platforms in conjunction with JMCIS'98 efforts;
- Contributing to the design of the interface between JMCIS and other tactical and non-tactical systems;
- Producing security strategy that provides guidance and documents preventative measures relating to computer security in development and operation;
- Ensuring JMCIS'98 complies with the DII COE security checklist

(as outlined in JMCIS'98 Security Policy and Requirements);

- Developing security testing criteria and incorporate within test plans;
- Developing the capability to monitor, detect, report and prevent network intrusion attempts;
- Monitoring security advisories published by the Computer Emergency Response Team (CERT); notify the JMCIS Program Manager (PM), Deputy PM and System Engineer when action is required; recommend courses of action to take in:
  - determining whether security breaches may have occurred,
  - initiating corrective action against security vulnerabilities,
  - mitigating risks to information security. [Ref. 2]

B. SUMMARY

JMCIS'98 represents a significant departure from the 'traditional' manner of providing C4I in the Navy. JMCIS'98 will incorporate many of the latest trends in

computing and information sharing; from Windows NT based architecture to usage of COTS products. JMCIS'98 will also rely more on WAN technology, specifically the SIPRNET and JWICS. These evolutionary moves from a traditional UNIX environment to one in which PCs and Web technology are center stage in the JMCIS architecture will ensure that Commanders are able to communicate efficiently and effectively with their forces, regardless of location. Security is an important consideration in the move to the JMCIS'98 architecture. The move to PCs and Web-based technologies open up many avenues of security problems that users of the Internet face every day. JMCIS'98 is approaching these problems by making security a life-cycle issue to be taken into consideration in every stage of planning and implementation.

III. JAVA

A major part of the JMCIS98 re-engineering effort revolves around the use of the Java programming language. Much of the functionality of JMCIS'98 will be implemented using Java applets and applications. This chapter focuses on the Java language and its security features.

Java is widely seen as the solution to many of the most persistent problems in client/server computing and on the World Wide Web. Java applets can easily be written and distributed across the World Wide Web, with or without direct user interaction. Java allows developers the flexibility to write one applet or application which can be distributed across a heterogeneous computing environment with little to no native code. This code is 'guaranteed' to run on any computer as Java and the Java Virtual Machine are resident. This is often referred to as 'Write Once/Run Anywhere.'

Java's ability to allow a central server to distribute cross-platform executable content across a heterogeneous computing environment can be viewed a central building block of implementing a 'thin-client/fat-server' system. In such a system, the client does not need or want to have applications resident. Instead it will download the executable when the service is required by a user. This setup has many advantages: configuration management, version control, allowing use of diskless workstations, etc..

The ability to field Write Once/Run Anywhere executables, allows Java programs written on one type of hardware or operating system to run unmodified on almost any other type of computer. Thus, portability is a major advantage for Java. Java provides its Write Once/Run Anywhere capability through the Java Virtual Machine. The Java Virtual Machine is implemented on top of a machine's native operating system. Java applications run on top of the virtual machine. The virtual machine insulates the application from differences between underlying operating systems and hardware.

A. JAVA SECURITY RISKS

The ability to distribute executables automatically over the network raises security concerns. Java addresses the problem by distinguishing applications from *applets.* Figure 3-1 illustrates the Java Security Reference Model. Java programs can exist in two forms: as applets, which travel across the Internet or intranet as part of a web page and run inside of the end-user's browser, or as traditional stand-alone applications. Figure 3-1 shows the relationships between Java applets and applications and the Java Virtual Machine. One of the biggest security issues relating to Java is what should a Java applet or application be allowed to do to the local machine. The differences between an applet and application and the privileges that are afforded each under the Java Virtual Machine are discussed next.

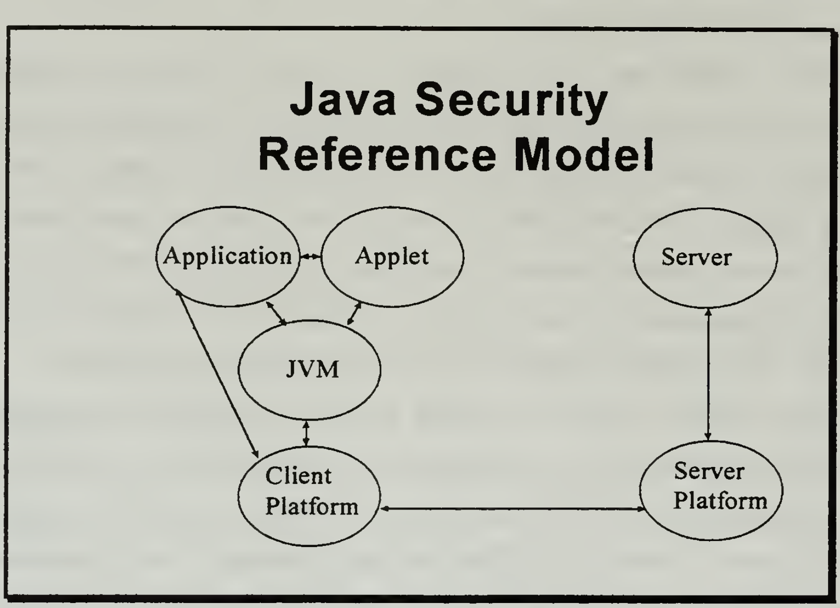


Figure 3-1. Java Security Reference Model [From Ref. 4]

1. Java Applications

Java applications add no new security concerns. Security comes from maintaining physical control over the systems, preventing end-users from downloading untrusted applications from the net, using virus checkers and other traditional security measures.

2. Java Applets

Java applets are subject to special Java 'sandbox' security restrictions due to the fact that they are regarded as 'untrusted' code which is generally downloaded over the World Wide Web and run on the local host machine. Applets pose the most insidious security threat since users often are unaware they are executing them though web browsing and applets have the capability to run behind the organization's firewall.

Applets are Java's form of executable content. They are a major security concern. Applets are small pieces of executable code which may be included in Web pages and run inside of the user's browser. While applets solve many of the important problems in client/server and network-centric computing, they also raise new concerns about security. Traditionally, organizations could protect themselves by controlling physical and network access to their computers and establishing policies for what kinds of software can be used on their machines. These steps include building a firewall between the Internet and the organization's intranet, obtaining software only from known and trusted sources, and using anti-virus programs to check all new software.

Use of applets potentially adds a new security vulnerability. A user searching an external Web site for information might inadvertently load and execute an applet without being aware that the site contains executable code. This automatic distribution of executables makes it very likely that software will be obtained from untrusted third parties. Since the applet is imported into the user's web browser and runs locally, this software could potentially steal or damage information stored in the user's machine on a network file server. Also, since this software is already behind the organization's firewall, the applet could attack other unprotected machines on the organization's intranet. These attacks would not be stopped by traditional security measures.

Java protects its users from these dangers by placing strict limits on applets. Applets cannot read from or write to the local disk. Stand-alone windows created by applets are clearly labeled as being owned by untrusted software. These limits prevent malicious applets from stealing information, spreading viruses, or acting as Trojan horses. Applets are also prohibited from making network connections to other computers on the network. This prevents malicious applets from exploiting security flaws that might exist behind the firewall or in the underlying operating system. While Java is not the first or only platform that claims to be a secure means of distributing executable code over the Internet, it is perhaps the best known and most widely used.

Java's security mechanisms are designed to allow a user to import and run applets from the Web or an intranet without damaging the user's machine. The applet's actions are restricted to its "sandbox." The applet may do anything it wants within the sandbox, but cannot read or alter any data outside of its sandbox. The sandbox model's purpose is to run untrusted code in a trusted environment so that if a user accidentally imports a hostile applet, that applet cannot damage the local machine.

Java automatically confines applets to the sandbox. End-users do not have to take any special action in order to ensure the safety of the machine. Because the sandbox prevents the actions required to spread a virus or steal information, instead of trying to identify a virus-infected executable or potential attacker, the sandbox does not require periodic updates with new virus detection mechanisms.

As mentioned above, the actions an applet is allowed to perform is restricted by the Java security mechanism. More specifically, if an applet has been loaded across the network, it is not allowed to:

- read files on the client file system
- write files to the client file system
- delete files on the client file system, either by using the **File.delete()** method or by calling system-level **rm** or **del** commands
- rename files on the client file system, either by using the **File.renameTo()** method or by calling the system-level **mv** or **mkdir** commands
- create a directory on the client file system, using either by using the **File.mkdirs()** methods, or by calling the system-level **mkdir** command
- list the contents of a directory
- check to see whether a file exists
- obtain information about a file, including size, type, and modification time stamp
- create a network connection to any computer other than the host from which it originated
- listen for or accept network connections on any port on the client system
- create a top-level window without an *untrusted window* banner
- obtain the user's username or home directory name through any means including trying to read the system properties: **user.name**, **user.home**, **user.dir**, **java.home**, and **java.class.path**
- define any system properties
- run any program on the client system using the **Runtime.exec()** methods
- load dynamic libraries on the client system using the **load()** or **loadLibrary()** methods of the **Runtime** or **System** classes
- create or manipulate any thread that is not part of the same **ThreadGroup** as the applet
- create a **ClassLoader**

- create a **SecurityManager**
- specify any network control functions, including **ContentHandlerFactory**, **SocketImplFactory**, or **URLStreamHandlerFactory**
- define classes that are part of packages on the client system
  [Ref 3:pp 35-36]

B. JAVA SECURITY ARCHITECTURE

Java's Security Architecture is composed of several different systems operating together. These systems range from security managers running inside of the application which imported the applet, to safety features built into the Java language and the virtual machine. The Java Sandbox security model can be described as a 'three-pronged attack.' This three-pronged attack is made up of the:

- Class Loader
- Class Verifier
- Security Manager

The term 'three-pronged' is used as the security model does not describe layers of protection. A breach of one of the prongs leaves the system vulnerable. [Ref 3:p. 36] We now take a close look at the Class Loader, Class Verifier and the Security Manager more closely.

1. Class Loader

When an applet is to be imported from the network, the web browser calls the applet class loader. The class loader is the first link in the security chain. In addition to fetching an applet's executable code from the network, the class loader enforces the name space hierarchy. A name space controls what other portions of the Java Virtual Machine an applet can access. By maintaining a separate name space for trusted code which was loaded from the local disk, the class loader prevents untrusted applets from gaining access to more privileged, trusted parts of the system.

Applets downloaded from the net cannot create their own class loaders. Downloaded applets are also prevented from invoking methods in the system's class loader.

2. Class Verifier

Before running a newly imported applet, the class loader invokes the verifier. The verifier checks that the applet conforms to the Java language specification and that there are no violations of the Java language rules or name space restrictions. The verifier also checks for common violations of memory management, like stack underflows or overflows, and illegal data type casts, which could allow a hostile applet to corrupt part of the security mechanism or to replace part of the system with its own code. Figure 3-2 illustrates how the class loader and verifier fall into Java's security hierarchy.

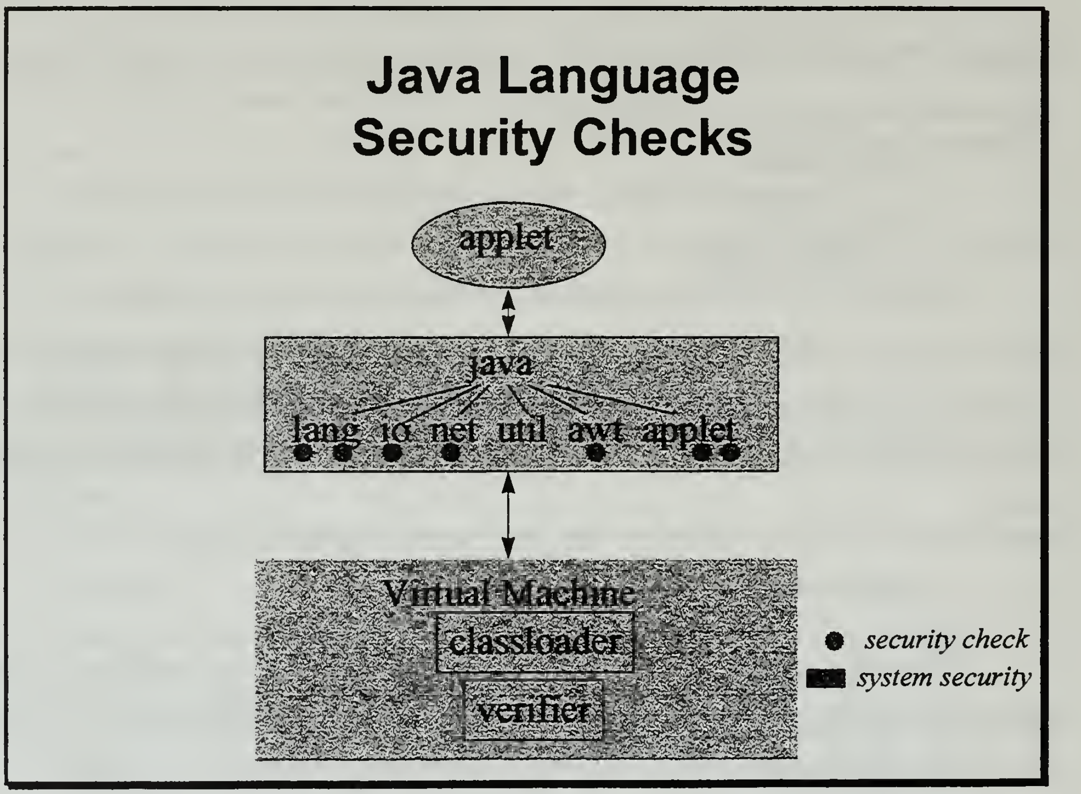


Figure 3-2. Java Language Security Checks [From Ref. 4]

3. Security Manager

The security manager defines the boundaries of the sandbox. The **SecurityManager** class in Java implements a policy for the execution of untrusted code. Normal applications do not use the security manager. The security manager is typically only used by Web browsers, applet viewers, and other programs that need to run untrusted code in a controlled environment. [Ref 5:p 469] The default Security Manager is quite restrictive, but methods can be overridden to allow more permissions if desired.

Whenever an applet tries to perform an action which could corrupt the local machine or access information, the Java Virtual Machine first asks the security manager if this action can be performed safely. If the security manager approves the action - for example, a trusted applet from the local disk may be trying to read the disk, or an imported untrusted applet may be trying to connect back to its home server - the virtual machine will then perform the action. Otherwise, the virtual machine raises a security exception and writes an error to the Java console.

The security manager will not allow an untrusted applet to read or write to a file, delete a file, get any information about a file, execute operating system commands or native code, load a library, or establish a network connection to any machine other than the applet's home server.

An application or a web browser can only have one security manager. This assures that all access checks are made by a single security manager enforcing a single security policy. The security manager is loaded at start-up and once it is loaded it cannot be extended, overridden or replaced. For obvious reasons, applets can not create their own security managers.

C. SUMMARY

In any discussion of computer security and threats, it is well worth the time and effort to identify any threats. In the case of Java programming for networked environments, the security concerns can be classified as:

- System Modification - Java has strong defenses against this type of attack.
- Invasion of Privacy - Java has strong defenses against this type of attack.
- Denial of Service - Java has weak defenses against this type of attack.
- Antagonism - Java has weak defenses against this type of attack.
  [Ref. 3:p. 30]

It is clear from the above list that the design of Java's security mechanisms was intended to combat attacks that would modify the system and invade privacy. It is important to note that individual organizations might want to strengthen or supplement

Java's defenses against attacks. For example, a combat system must be able to survive denial of service attacks as it must be available at all times.

Java's Developers made great efforts towards security. They recognized the potential hazards of executing code that was downloaded from a network. The sandbox was designed to contain Java applications to prevent damage (to the greatest extent possible) from malicious code. However, restricting Java applets to the sandbox eliminates much of the potential functionality. With the fielding of Java Developer's Kit 1.1 to replace version 1.0.2, Sun has made great strides to increasing the potential functionality of Java applets without sacrificing the security of the sandbox model. These changes to Java will be discussed in the next chapter.

IV. JAVA CRYPTOGRAPHY ARCHITECTURE AND EXTENSION

In JDK 1.1, Sun Microsystems took steps to open up the functionality of applets, without sacrificing the security of the sandbox model, by incorporating cryptographic features to allow for 'trusting' an applet. This chapter will discuss this feature, as well as outlining the changes to the language that make up the Java Cryptography Architecture and the Java Cryptography Extension (JCE 1.1).

A. JAVA'S NEW SECURITY FACILITIES

The Java Security Architecture goes a long way towards protecting the end-user's machine and networked computing resources from damage or theft by a malicious applet. Users can usually run untrusted code obtained from the network without harm to their system.

The Java Security Architecture does not address several other security and privacy issues. Authentication can be used to guarantee that an applet comes from the place it claims to have come from. Digitally-signed and authenticated applets can be promoted to the status of trusted applets, and then allowed to run with fewer security restrictions. Encryption can ensure the privacy of data passed between an applet client and a server on the Internet. The newest version of Java, contained in the Java Developer's Kit (JDK) 1.1, does much to extend Java's security model in each of these areas. Java's new security features include:

- Signed JAR Files
- Java Cryptography Architecture
- Java Cryptography Extension

1. Signed JAR files

All networked systems are potentially vulnerable to downloading and executing malicious code. Java's designers took this into account when designing the Java Security Architecture. Since the Security Architecture greatly restricts the activities of an applet, as discussed in Chapter III, users could usually rely on the Java Security Architecture to protect their system from malicious code. Java's designers also saw great utility in allowing an applet to operate outside the security architecture. An applet that was afforded the privileges of an application would be more useful in many networking environments. A prime example of this would be for a user to download an applet and have that applet access files on the local machine or make connections to another server on the user's behalf, as is needed in the JMCIS'98 network architecture. An applet that was trusted to carry out these activities without harm is quite useful.

"Signed applets" give us the same level of confidence in network distributed software as shrink wrap does to commercial software today. To sign an applet, the producer first bundles all the Java code and related files into a single file called a Java Archive, or JAR. The producer then creates a string called a digital signature based on the contents of the JAR.

JAR files also help to alleviate another common networking problem. Currently, many Java applets take a very long time to download and begin running. The problem is that current Internet protocols move web pages across the Internet one file at a time. Since there is overhead associated with each request for a file, web pages and Java applets which are composed of many small files might spend more time requesting those files and waiting for replies than they spend actually moving the information. Since a JAR file bundles all the information needed by the applet and its web page into a single file, the entire page can be downloaded with a single request. For many web pages, this will greatly reduce download times.

JAR files and digital signatures can also be used for Java applications. While Java applications are more trustworthy than applets because they do not travel over the Internet and are subject to an organization's traditional security policies, applications are subject to several types of attack. For example, viruses spread by modifying existing applications to include a copy of the virus. Since a virus would not be able to produce a valid digital signature for the altered program, the Java system could detect that a signed application has been tampered with, and refuse to run it. Since the JAR signature system will work with all types of information, not just Java files, JAR signatures can also be used to protect data files and other information.

Since digital signatures allow us to assign to Java applets the same level of trust which we assign to shrink-wrapped applications, it may be useful to relax the Java security restrictions for some applets. Signed applets can be used to create such an environment. If the end-user has previously told the Java system that a particular web publisher is trusted and a signed applet from that publisher has arrived over the Internet and has been authenticated, then the Java Security Manager could allow that applet out of the sandbox, and treat it as an application.

The Security Manager could also enforce different levels of control based on how much a particular publisher is trusted, or on how much the Network as a whole is trusted. A very security-conscious user could configure the system to allow signed applets to run only inside the sandbox, and to prevent any unsigned applet from running at all. Another user could configure the system to allow the applet to access only one particular directory on the hard disk.

2. Java Cryptography Architecture

While the sandbox and signed applets can protect against malicious applets, information traveling between the applet and a server on most networks is still vulnerable to theft. This is because the network itself is an insecure medium, this may still be the case in a "system high" network such as SIPRNET or JWICS in that there may be authorized network users who are not authorized JMCIS'98 users. An attacker attached to a central portion of the network can read all information which travels through that portion of the network. To secure against this type of attack, all information flowing between the applet and its server must be encrypted.

Java encryption facilities have recently been released and others are being developed. These facilities will allow applet developers to select the type of encryption algorithm used, to negotiate with the server to create the keys used in the encryption and to do the actual encryption of the data.

The Java Cryptography Architecture (JCA) is the framework for accessing and developing cryptographic functionality for the Java Platform. It encompasses the parts of the JDK 1.1 Java Security API related to cryptography (currently, nearly the entire API), as well as a set of conventions and specifications provided in this document. It introduces Sun's default "provider" and the provider-architecture that allow for multiple and interoperable cryptography implementations.

a. Java Security API

The Java Security Application Programming Interface (API) is built around the **java.security** package and its subpackages. The first release of the Security

API primarily includes support for digital signatures. This first release of Java Security (in JDK1.1) has APIs for:

- Digital Signatures
  DSA
- Message Digests
  MD5
  SHA-1
- Key Management
- Access Control Lists
  [Refs. 6 & 7]

The structure of the Security API is designed to allow implementation independence and interoperability, as well as algorithm independence and extensibility. Implementation independence and algorithm independence aim to let users of the API utilize cryptographic concepts, such as digital signatures and message digests, without concern for the implementations or even the algorithms being used to implement these concepts.

Implementation independence is achieved using a "provider-based" architecture. Applications may simply request a particular type of object, and get an implementation from an installed provider. If desired, an application may instead request an implementation from a specific provider. When implementation independence is not desirable, developers can indicate the specific implementations they require.

Algorithm independence is achieved by defining types of cryptographic "engines" (algorithms), and defining classes that provide the functionality of these cryptographic engines. These classes are referred to as engine classes, for example, the *MessageDigest* and *Signature* classes.

Several methods of the *MessageDigest* and *Signature* classes are used in Chapter V. The *Signature* class is used in the creation and verification of digital signatures. Signature objects may have one of three states: unitialized, sign and verfiy. We will use signature objects in implementing our protocols. A signature object is unititialized at creation. The object can be initialized (prepared) for signing or verifying by using either the *initSign* or *initVerify* methods. The object can be used to digitally sign using the update method which supplies the data to be signed to the signature object and then calling the *sign* method to create the signature. The signature object is also used to verify a signature by initializing the signature object for verification by calling *initVerify* which prepares the object for verification. The data to be verified can then be supplied to the signature object by calling *update* and then a call to *verify* for verification.

b. Cryptography Package Providers

Java Security is essentially an implementation independent layer. It introduces the notion of a Cryptography Package Provider (referred to as a "provider"). This term refers to a package (or a set of packages) providing a concrete implementation of a subset of the cryptography aspects of the Java Security API.

JDK1.1 comes standard with a default provider named "SUN". New providers can be added either:

- Statically: An authorized administrator or user can add a provider to the persistent list of providers that are available to all programs.
- Dynamically: An authorized program can add a provider at runtime for its private or temporary use.

Java provides a set of APIs allowing users to query which providers are installed.

As mentioned above, the default provider in JDK1.1 is called "SUN". This provider contains:

- Digital Signature Algorithm (NIST FIPS 186)
- MD5 (RFC 1321)
- SHA-1 (NIST FIPS 180-1)

"SUN" is the highest-priority (default) provider that provides the default implementation. Preference order can be changed by the user, i.e., it is configurable. The preference order is the order in which providers are searched for requested algorithms when no specific provider is requested. [Ref. 7:p. 4] "SUN" also supplies a simple system key and trust management mechanism, including a persistent database of principals, keys, and X.509.v1 certificates. The ***javakey*** tool interfaces with this persistent database using a set of Java supplied APIs.

3. Java Cryptography Extension

The Java Cryptography Extension (JCE) version 1.1 is a set of APIs for cryptographic functionality, including symmetric, asymmetric, stream and block encryption as well as key generation and management. There are also implementations in Java of a subset of the APIs. The JCE supplements the functionality of the JCA. Together, the JCE and the JCA provide a complete, platform-independent cryptography API. The JCE is provided as a separate release due to United States exportation laws. The JCE uses the same provider architecture as the JCA does.

The JCE provides APIs for:

- Symmetric bulk encryption
  DES
  3DES (Triple DES)
  RC2
  IDEA
- Symmetric stream encryption

RC4

- Asymmetric encryption

RSA

- Built-in extensible multiple modes

Electronic Codebook Mode (ECB)
Cipher Block Chaining (CBC)
Cipher FeedBack Mode (CFB)

- Padding

Public Key Cryptography Standard (PKCS) #5
Privacy Enhanced Mail (PEM) style padding
[Refs. 8 & 9]

The JCE also includes implementations of the following algorithms which extend the SUN provider:

- DES
- 3DES
- ECB and CBC modes
- PKCS#5-style padding

[Refs. 8 & 9]

Note: Currently there is not an implementation of the RSA asymmetric key cryptography algorithm in JCE 1.1 due to licensing restrictions from RSA.

Several of the JCE classes are used in implementing the Trusted Code and the Secure Password Transmission Protocols discussed in Chapter V. Among them is class *Cipher*. The *Cipher* class defines behavior for encryption and decryption, including initialization as well as the encryption or decryption operation [Ref. 9:p. 2]. A cipher object is used to carry out the encryption and decryption processes. Methods pertaining to a cipher object include the creation, initializing and the encryption and decryption processes. A cipher object is created by making a call to the appropriate cryptographic algorithm using the *getInstance* method. The newly created cipher object is unitialized. The object can be initialized to encrypt or decrypt data by calls to the *initEncrypt* or *initDecrypt* methods, respectively. Once the object is initialized for encryption or decryption a call to the *crypt* method will carry out the intended operation.

The *CipherInputStream* is also used by our protocol implementations. The purpose of *CipherInputStream* is to encrypt or decrypt the data that is passing through it. Typically, this stream would be used as a filter to read an encrypted file. [Ref. 9:p.4]

B. SUMMARY

The Java Cryptography Architecture and the Java Cryptography Extension are recent additions to the Java language and Java Developer's Kit 1.1 that add cryptographic functionality. The JCA and JCE provide APIs for common cryptographic algorithms. These cryptographic additions to the JDK allow for code signing and data encryption.

JDK 1.1 also allows for a trusted applet to enjoy the privileges of a local application by using digital signatures and JAR files. A full implementation of this system allows fine-grain tuning on what system privileges a particular applet is allowed to use.

Java security continues to evolve to allow for more user-definable and robust security measures. The security additions to the Java Developer's Kit 1.1 can be used to implement protocols verifying trusted code and transmitting passwords securely. This is the subject of Chapter V.

V. JMCIS'98 SECURITY PROTOCOLS

The JMCIS'98 Network Architecture is composed of clients, web servers, gateway servers, access servers and databases. A client connects to a web server and downloads executable content to accomplish a task. This executable content may need to connect the client to a gateway server. The client sends database queries to the gateway server which then forwards them to an appropriate access server. The access server responds to queries by accessing a database. Query replies are then sent back to the client via the gateway server. This architecture is illustrated in Figure 5-1 below.

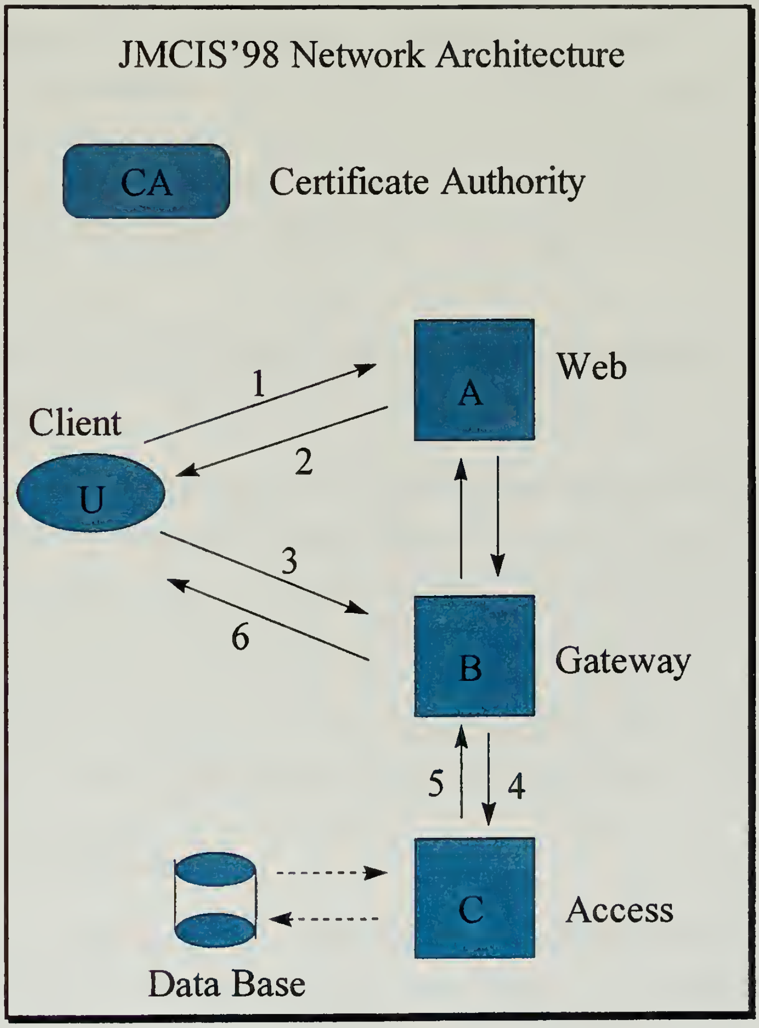


Figure 5-1: JMCIS'98 Network Protocol

This chapter introduces and describes two protocols: the Trusted Code and the Secure Password Transmission Protocol which were developed by Professor Volpano in coordination with Space and Naval Warfare Systems Command JMCIS'98 engineers. The former allows clients to download executable content, from a web server and decide whether it can be trusted based on signature verification. The latter allows a client to securely transmit passwords to a gateway server. Implementations of these two protocols using features of Java's JDK 1.1, the Java Cryptography Architecture and the Java Cryptography Extension will be described. We also describe Netscape's Secure Sockets Layer technology and describe how it might be used in the JMCIS'98 Network Architecture in place of the Secure Password Transmission protocol.

Section A of this chapter will discuss the Trusted Code Protocol and its implementation using the Java Cryptography Architecture. Section B examines an alternative approach using features of Java's JDK 1.1, specifically, signed JAR files and the ***javakey*** database. Section C examines the Secure Password Transmission Protocol and its implementation using the Java Cryptography Extension. Section D is a discussion of SSL and its use in JMCIS'98 as an alternative to the Secure Password Transmission Protocol.

A. TRUSTED CODE AND SECURE PASSWORD TRANSMISSION PROTOCOLS

This section will present the Trusted Code and the Secure Password Transmission Protocols, as developed by Professor Volpano [Ref. 11]. The following sections will discuss how the Trusted Code and the Secure Password Transmission Protocols can be implemented using the Java Cryptography Architecture and the Java Cryptography Extension. An alternative to the Trusted Code Protocol using the Java JDK 1.1 functionality as well as an alternative to the Secure Password Transmission Protocol using Netscape's Secure Sockets Layer technology are also discussed.

1. Trusted Code Protocol

The Trusted Code Protocol is intended to allow a client to download executable content across the network and then establish trust in that code based on a digital signature. The steps of this protocol are as follows:

1. A client receives a class file as a signed entity from a web server. It extracts the code signer's name *N* and checks whether it trusts *N* by looking for *N* in a local table. The class file is rejected in *N* is not found.

2. Next, the client attempts to verify that *N* actually signed the code. In order to do this it requests from the server a code-signer identity object for *N* (a certificate for *N*) which is purportedly signed by some CA, say *C*.

3. Upon receipt of the certificate, it extracts the name of the CA contained in the certificate, in this case *C*, and gets the public key for *C* that is held locally. It verifies the certificate using *C*'s public key. This confirms that the public key in the certificate is the public key of the name, say *N′*, found in that certificate.

4. If *N*=*N′*, it attempts to use the public key in the certificate to verify the signature of the class file. If successful, the client knows that *N* signed the class file, and since it trusts *N*, it executes the class file.

2. Secure Password Transmission Protocol

The Secure Password Transmission Protocol is concerned with the confidential transmission of a password from a client to a server and the secure transmission of information from that server back to the client. Unlike alternative approaches, the protocol is not an authentication protocol, for it does not assume that a server and client share a secret. The protocol does assume that every socket, which consists of an IP address and a port number, has a public and private key. The protocol behaves as follows:

1. A client requests from a server a public-key-signer identity object for the server's socket *S*. The object is a certificate for *S* purportedly signed by a CA, say *C*.

2. Upon receipt of the certificate, the client extracts the name *C* and retrieves what it believes locally to be *C*'s public key.

3. It verifies the certificate using *C*'s public key. This confirms that the public key in the certificate is the public key of the socket, say *S'*, found in that certificate. If *S*=*S'*, then the client generates a one-time symmetric key *k* and sends *k*, its password and its name to socket *S*, bundled and encrypted together using the public key found in the verified certificate.

4. The server at socket *S* attempts to decrypt the bundle. If successful, it services some request with the supplied password and then sends the results back to the client encrypted using the supplied key *k*.

In the final step, we say the server at socket *S attempts* to decrypt the bundle. The idea is that a trusted server at socket *S* knows the private key for *S*, an untrusted server does not. Thus an untrusted server at socket *S* must either send unverifiable socket certificates for *S* to clients (by taking a valid signature for *S* and associating with it, its own public key), or send a verifiable certificate, yet be unable to decrypt the bundle from the client.

The one-time symmetric key also serves as a pad to prevent replays of an encrypted password, just as a challenge does for a secret exchanged in the Point-to-Point Protocol's (PPP's) Challenge Handshake Authentication Protocol (CHAP).

Before describing implementations of these protocols, we need to address implementing the cryptographic keys they require. To assist in this, we will make use of a utility package created for JMCIS'98. [Ref. 11] This package contains the Java methods ***CreateKeys***, ***SignBytes***, ***CodeSignerID***, ***RunApp***, ***SocketID***, ***ConnectClient*** and ***ServerSide***. These methods will be described as they are encountered in the implementations of the protocols.

3. Implementing Keys

Part of implementing the Trusted Code and the Secure Password Transmission Protocols is implementing the cryptographic keys they require. The protocols require the use of a Certification Authority (CA) who will issue certificates verifying the trustworthiness of other entities, such as code signers, and server-side certificates for use by the web and gateway servers.

We describe how the two protocols can be implemented in a network environment which includes a CA (a certificate authority), a code signer entity, named BigNavy, and a gateway server, named Otter. We will be using the ***CreateKeys*** method from the JMCIS'98 utilities package. It is used to create asymmetric keys for the Digital Signatures Algorithm (DSA). [Ref. 7:p. 3 and Ref. 11]

First we create the asymmetric keys (public and private keys) for the CA and BigNavy. Otter will be dealt with separately later in this chapter. For each identity there will be associated *.PublicKey* and *.PrivateKey* files, e.g., *BigNavy.PublicKey* and *BigNavy.PrivateKey*. ***CreateKeys*** creates a DSA public and private key for the entity given as an argument.

As an example, ***CreateKeys*** generates the DSA key pair for BigNavy by the call:

```
java CreateKeys BigNavy
```

This call creates two files; *BigNavy.Private* to store the BigNavy's private key and *BigNavy.Public* to store the BigNavy's public key. Here is how the JCA is used in ***CreateKeys***:

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
keyPairGen.initialize(1024, new SecureRandom());
KeyPair BigNavy = keyPairGen.generateKeyPair();
```

This code is creating an instance of the key pair generating class for the DSA algorithm. In this example a 1024-bit public key is generated. Once the object is instantiated, it is initialized with the key length and a random number generator. This random number generator is provided by Java's *SecureRandom* class. The *SecureRandom* class is intended to provide a software-based, platform independent, good-quality random number generator. [Ref. 7:p. 16] The key pair generating object is then directed to generate the key pair and save them (the .Public and .Private key files) in association with the BigNavy entity.

4. Implementing Certificates

Certificates are now implemented for the CA, BigNavy and Otter by using Java's *Identity* class. The identities are created in the form of *Identity* class objects, which are abstract and associate a name with a public key. The name contained in the identity is immutable in that it cannot be changed, ensuring a one-to-one mapping between keys and identities [Ref. 7:p. 14]. This is important in preventing malicious tampering with the identity. A code signer identity is created for BigNavy using the ***CodeSignerID*** method included in the JMCIS'98 utility package.

```
java CodeSignerID BigNavy BigNavy.PublicKey
```

In this example, the code signer identity is created for BigNavy. A code signer identity is a serialized identity object that associates a name with a public key. The BigNavy identity is a code signer and is used to sign class files. It is represented by the file:

BigNavy.Identity

Now we an have identity created for BigNavy. The next step is for the CA to endorse the association by signing it in the form of what is usually referred to as a certificate. The CA signs identities using its private DSA key. Recipients of the certificate may verify it using the CA's public key. Certificates in our case are merely signed byte streams, which bypasses issues of certificate format (ASN.1/DER), various X.509 versions and other standards.

In order to generate certificates we use the ***SignBytes*** utility which is part of the utility package designed for JMCIS'98. ***SignBytes*** creates a signed byte stream certificate by signing an identity with the CA's private key.

```
java SignBytes BigNavy.Identity CA
```

The CA only signs identities (.Identity files). The code signer, in this case BigNavy, will sign the class files. We now have a signed identity in a file that ends with *.sig*:

BigNavy.Identity.sig

It serves as a "certificate" (signed identity) for BigNavy.

B. TRUSTED CODE PROTOCOL IMPLEMENTATION USING JAVA'S JCA

The Java Cryptography Architecture provides the means to digitally sign code and to verify the authenticity of these signatures. This JCA functionality can be used to implement the Trusted Code Protocol.

The CA and the BigNavy code-signer have certificates. In implementing the Trusted Code protocol using the JCA, the CA identity-signing certificate and the BigNavy code-signing certificate are held on the web server. As an example of using the JCA functionality, assume that navy programmers write and compile an application called ***HelloWorld.java***. This application is then signed by the BigNavy code-signer to verify that it is trustworthy code.

```
javac HelloWorld.java
java SignBytes HelloWorld.class BigNavy
```

It is important to note here that a signature included in a certificate is computed over a name and a public key. An assumption in this signing process is that only an authorized code-signer (authorized to represent code-signer BigNavy) performs the signing step after confirming the code is correct. The JCA calls that would make up the method ***SignBytes***, which is included in the JMCIS'98 utility package [Ref. 11], are included below as indicative of the type of code that would be used. Using the DSA implementation included in the JCA , we have:

```
Signature dsa = Signature.getInstance("DSA");
dsa.initSign(BigNavy.getPrivate()):
dsa.update(HelloWorld.class);
byte[] SIG = dsa.sign();
```

Not shown here is that ***SignBytes*** produces the file *HelloWorld.class*.sig that contains the signature *SIG* and the class file *HelloWorld.class*.

On the client side, a signed file is downloaded by a client by accessing a web page. The client's intentions are to execute this code if it can be verified to be trustworthy. The Java application ***RunApp*** is part of the JMCIS'98 utilities package [Ref. 11]. ***RunApp*** implements the Trusted Code Protocol. The following call to ***RunApp*** would run the *HelloWorld.class* file if the protocol succeeds.

```
java RunApp HelloWorld.class.sig
```

An example of the some JCA calls included in ***RunApp*** is given below.

```
Signature dsa = Signature.getInstance("DSA");
dsa.initVerify(BigNavy.getPublic()):
dsa.update(HelloWorld.class);
boolean verifies = dsa.verify(SIG);
```

Here "*SIG*" is extracted from the signed class file *HelloWorld.class.sig*. An assumption that we have made in considering Java's new code-signing features for inclusion in the JMICIS'98 system is that future versions of commercial browsers will implement these features of the Java Cryptography Architecture.

C. AN ALTERNATIVE APPROACH TO VERIFYING TRUSTED CODE USING JDK 1.1

Java's JDK 1.1 provides an alternative to verifying trusted code using signatures. The JDK 1.1 approach is to package up application or applet (.class) files into a Java Archive (JAR) file (described in Chapter III). The JAR system was primarily designed to allow for a method of establishing trust in Java applets downloaded across a network and allowing these applets to operate outside of the sandbox model.

A JAR file is signed by a 'signer authority,' which combines the functionality of the CA identity-signer and the BigNavy code-signer from above. The client trusts the SA and thus trusts the code that is signed by the SA. The client will then be able to allow the applet to execute outside of the sandbox, to include reading and writing to the local (client) file system and making connections to another server, such as a gateway server.

The JDK 1.1 approach uses a *certificate directive file*. The purpose of this file is to give the parameters to be included in the certificate and the file in which the certificate will be stored. The following data are included in a typical certificate directive file.

- The identity of the signer

  issuer.name=BigNavy

- The certification level to use for the signing (only level 1 is currently supported)

  issuer.cert=1

- The identification of the subject

```
subject.name=BigNavy
```

- The components of the X500 name for the subject

```
subject.real.name=BigNavy
subject.org.unit=Navy
subject.org=United States
```

- Other parameters: start and end dates for validity and expiration of the certificate. Serial number. File in which the output certificate is placed.

```
start.date=25 Sep 1997
end.date=01 Jan 2000
serial.number=1001
out.file=BigNavy.x509
```

These certificates are required for use by ***javakey***. The ***javakey*** tool is a command-line interface to key and certificate generation and management tools in JDK 1.1. The ***javakey*** tool manages a system database of entities. Each entity may have public and private keys and/or certificates associated with it. Each entity may be declared to be trusted or not. Any entity in the database may be an "identity," which has a public key associated with it, or a "signer," which has both a public and private key and can sign files. The "identity" in this case should not be confused with the "identity" in the JCA implementation of the Trusted Code Protocol. An identity in this case is simply an entity that has been deemed to be trusted. On the client side, the SA would be an identity and the client would hold the public key of the SA. This would allow the client to verify the signed files received from the SA. An identity in the JCA approach can either sign code or identities.

To implement the Trusted Code Protocol using the JDK 1.1 JAR file functionality, we need to accomplish two steps, first to establish a trusted identity and, secondly to download the signed JAR file and verify that it is from a trusted identity.

1. Establishing a Trusted Entity

In order for the applet to be trusted, we must establish a code signer that will certify the code and sign it. This approach is similar to that taken in the JCA implementation. In the JCA approach, the CA trusted the code-signer and the code-signer trusted the code. Since the client trusted the CA, the client could safely execute code that was certified in this manner. In the JAR file approach, the code-signer trusts the code and the client trusts the code-signer. Therefore the client can trust the code.

In the JAR file approach, a code-signing entity must be trusted by the client. Suppose that this entity is "BigNavy." The steps that follow are those steps one must take using JDK 1.1. The code fragment to establish this trusted entity and the fragments that follow are intended to be only sketches of the code required to perform these actions.

- First create the BigNavy identity as a trusted entity in the identity database using ***javakey*** as described above

  ```
  javakey -cs BigNavy true
  ```

- Generate a DSA 512-bit key pair for BigNavy, and store the public key in a file named ***BigNavy_pub*** and the private key in a file named ***BigNavy_priv***

  ```
  javakey -gk Duke DSA 512 BigNavy_pub BigNavy_priv
  ```

- Generate an X.509 certificate for BigNavy, and store it in the file ***BigNavy_x509***. This output file name is given in the directive file named cert_directive_BigNavy

  ```
  javakey -gc cert_directive_BigNavy
  ```

- Create the Java Archive File

  ```
  jar cf signedNavyApplet.jar navyApplet.class navyApplet.html
  ```

- Sign the archive, using the parameters given in the ***sign_directive_BigNavy***

  ```
  javakey -gs sign_directive_Duke signedNavyApplet.jar
  ```

- Move the signed archive to a file suffixed in .jar.

  ```
  mv signedNavyApplet.jar.sig signedNavy.Applet.jar
  ```

- You can show the contents of the signed archive using

  jar tvf signedNavyApplet.jar

- Finally, you can show the contents of the identity database using

  javakey -ld

These steps illustrate sketches of the steps required to create the trusted identity in the identity database, create the JAR file containing the appropriate files and to sign the JAR file with the X.509 certificate of the trusted entity.

2. Downloading A JAR File

Once the applet is included in the signed JAR file, it is ready for retrieval. In a web environment, the only difference between downloading the signed JAR file and any other applet lies in the syntax of the hypertext markup language (HTML) tag. Below are examples of a regular HTML tag and how the HTML tag would be constructed for our sample trusted applet.

- Normal HTML Tag for a *.class* file

  <applet code=navyApplet.class" width=500 height=50> </applet>

- HTML Tag for a JAR file

  <applet code=navyApplet.class archive="signedNavyApplet.jar" width=500 height=50> </applet>

The only difference in the HTML tags is the inclusion of the archive name in the tag for downloading a JAR file.

Currently, only the Java applet viewer supports verification of signatures using JDK 1.1. This functionality could, and probably will, be introduced in future versions of the most popular browsers, such as Netscape Navigator and Microsoft Internet Explorer.

As mentioned above, the result of creating a trusted entity and having that entity sign the executable content code, in this example a Java applet, is that the client can download this code and execute it without the sandbox restrictions. The client is now able to use this applet to make connections to servers other than the applet's origin server. This has allowed us to successfully download an applet to the client from the server and have the client safely execute this code, as well as to allow the applet to read from and write to the local file system. This is important in JMCIS'98 since an applet comes from a web server and then connects to a gateway server.

D. IMPLEMENTING THE SECURE PASSWORD TRANSMISSION PROTOCOL USING JAVA'S JCE

In the JMCIS'98 architecture, the job of the gateway server is to route requests and queries from a client to the appropriate access server. A necessary part of this section of the protocol entails the transmission of the user's password across the network to the gateway server. It is of utmost importance that the confidentiality of the information that is being transmitted between the client and the gateway server.

The following section will explore how the Java Cryptography Extension can be used to implement the Secure Password Transmission Protocol. Section E below discusses an alternative to secure password transmission using Netscape's Secure Sockets Layer technology.

As mentioned in Chapter IV, the Java Cryptography and the Java Cryptography Extension introduce a set of APIs for cryptographic functionality as well as implementations of DES and 3DES in ECB and CBC modes which extend the default "SUN" provider.

In order to implement the Secure Password Transmission Protocol we will use a socket identity for server Otter. The purpose of this socket identity is to uniquely identify the IP address and the port to which the server listens.

The Otter socket identity is created in the same way that the BigNavy code-signer identity was created earlier in this chapter. Now the Otter certificate will associate the name (IP address) and port the server listens to with a public key. In this case, the public key will be generated by the RSA algorithm vice DSA in the BigNavy example.

To create the socket identity, the Java application ***SocketID*** (part of the JMCIS'98 utilities package) uses the following call:

```
java SocketID otter.cs.1610 otter.cs.1610.PublicKey
```

This call to SocketID creates the Otter socket identity and stores it in file:

```
otter.cs.1610.Identity
```

To create the certificate, the Otter socket identity is signed with the private key of the CA by the following call.

```
java SignBytes otter.cs.1610.Identity CA
```

The result of this call is the Otter socket identity certificate which has been signed by the CA. This certificate is stored in the following (.sig) file.

```
otter.cs.1610.Identity.sig
```

We will use this certificate in the implementation of the Secure Password Transmission Protocol. This certificate resides at the gateway server.

The first step of the protocol implementation is for the client to make a connection to the gateway server and request the socket identity certificate for Otter. ***ConnectClient***, from the utility package, is used here to demonstrate how this could be accomplished. ***ConnectClient*** implements the Secure Password Transmission Protocol. First a connection is established between the client and the server, in this case the gateway server Otter. The client then requests to download a signed socket identity for the gateway server. The gateway server responds by supplying its socket ID certificate and the CA's certificate verifying the servers socket identity to the client. The client, upon receiving both certificates, extracts the name of the CA from the CA's certificate. The client will use the public key it holds for that CA to verify the certificate. If verification is successful, the client extracts the socket's public key from the CA's certificate. A query object consisting of the client's name, password, query and a newly generated one-time symmetric key are written to a file which is then encrypted using the socket's public key.

On the server side the method ***ServerSide***, from the utility package, simulates the steps the server takes in the protocol. The gateway server Otter responds to the client-originated query object by sending the results to the client, encrypted using the symmetric key supplied by the client. The server then discards the symmetric key.

This protocol can not currently be implemented in JCE 1.1 since it does not contain an implementation of an asymmetric cryptography algorithm, such as RSA. The JCE 1.1 does not currently provide an implementation of RSA in Java, due to licensing restrictions on RSA which prevent it from being implemented in Java. However, there is an RSA API provided in JCE 1.1. This API is refined somewhat in JCE 1.2 which is now available. Only the Data Encryption Standard (DES), a symmetric key cryptography algorithm along with its associated key generation methods, is implemented in JCE1.1.

The following section provides an example of how the Secure Password Transmission Protocol can be implemented using the JCE. This example uses RSA asymmetric key and DES symmetric key cryptography. The protocol will be implemented by having a client connect to a gateway server, retrieve the socket ID and the CA certificates from the gateway server. Following verification of the identity of the socket, the client will generate a symmetric key for the gateway server to use in encrypting data for transmission from the server to the client. The client will bundle up the client's name, password, the query and the symmetric session key and encrypt this bundle with the gateway server's public key obtained from the socket ID certificate. The gateway server will then decrypt the query bundle with its own private key. The session key will be used to pass the query results back to the client. The gateway server will then discard the symmetric key. It is important to point out that the client's query bundle is encrypted with the gateway server's public key, thus only the correct gateway server (Otter) is able to decrypt it. Without Otter's private key the query bundle is useless.

As an example, consider that the client has successfully connected to the gateway server and has asked for and retrieved the certificate for Otter. The client will use the CA's certificate to verify Otter, and Otter's certificate to verify the identity and obtain the public key of the socket. Having accomplished this, the client can then generate a private (symmetric) key to be as a "session" key for the gateway server to pass the query results back to the client. The following example illustrates the functionality of the JCE using the "SUN" provider. We will assume that the socket ID created in the introduction to this section consists of an RSA asymmetric key for purposes of this example. The next step is for the client to generate the key to be passed to the gateway server. A DES key is created and passed as part of the query object.

- Generate a key

```
SecureRandom random = new SecureRandom();
KeyGenerator keygen = KeyGenerator.getInstance("DES");
Key key = keygen.generateKey(random);
```

At this point the client has generated the symmetric key which will be included in the query object, along with the client's name, password and the query itself. The query bundle is encrypted using the gateway server's public key (RSA) and the entire encrypted bundle is passed to the gateway server.

The gateway server, upon receiving this query object, decrypts the package using its own private key. The gateway server will then pass the client's name, password and the query to the appropriate access server (refer to Figure 5-1) and will retain the symmetric key to pass the query results back to the client. Upon receiving the query results from the access server the gateway server will encrypt the results with the symmetric key generated by the client. The following JCE calls illustrate how a cipher object is created, initialized with a key and used for encryption.

- Create a cipher object

```
Cipher des = Cipher.getInstance(“DES”,“CBC”,“PKCS#5”);
```

- Initialize the cipher object for encryption

```
des.initEncrypt(key);
```

- Encrypt

```
byte[] cipherQuery = des.crypt(query);
```

Next the gateway server sends the encrypted query results to the client. Then the gateway server purges the symmetric key.

Upon receipt of the encrypted query results, the client will create its own cipher object, initialize it for decryption and decrypt the query results. This process is accomplished using the following calls.

- Create a cipher object

```
Cipher des = Cipher.getInstance(“DES”,“CBC”,“PKCS#5”);
```

- Initialize the cipher object for decryption

```
des.initDecrypt(key);
```

- Decrypt

```
byte[] decryptedQuery = des.crypt(cipherQuery);
```

Thus, the client has initiated a query paired with a symmetric key, passes the query to the gateway server encrypted with the gateway server's public key and received a query response encrypted with the symmetric key.

E. AN ALTERNATIVE APPROACH TO SECURE PASSWORD TRANSMISSION USING THE SECURE SOCKETS LAYER

An alternative to the Secure Password Transmission protocol described above is Netscape's Secure Sockets Layer (SSL) technology. SSL, offered by Netscape in their Secure Commerce Server, has gained wide spread use on the Internet as a means to secure Web transactions, including financial transactions.

The primary goal of the SSL Protocol is to provide reliable private communication between a server and client. SSL is composed of two layers, the Record layer and the Handshake layer. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP), is the SSL Record Protocol. The SSL Record Protocol is used for encapsulation of various higher level protocols. Starting with a message to be transmitted, the Record Protocol fragments the data into manageable blocks, optionally compresses the data, applies a Message Authentication Code (MAC), encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher-level clients.

The SSL Handshake Protocol allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives data. This negotiation process consists of the client stating which ciphers and key exchange algorithms (i.e., RSA, Diffie-Hellman, or Fortezza) it has in preferred order, the server checking to see if it has a compatible cipher and key exchange and then signaling an agreement on a particular cipher. The Key Exchange algorithm only is relevant when the server has no certificate, or a certificate used only for signing (i.e., DSS or sign-only RSA). The agreement that is reached is the SSL **CipherSpec**, the set of agreed upon ciphers in current use. The overall purpose of the Handshake protocol is to agree on a **CipherSpec**.

One advantage of SSL is that it is application protocol independent. A higher level protocol can be layered or implemented on top of the SSL Protocol transparently.

The SSL protocol provides connection security that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption.
- The peer's identity can be authenticated using asymmetric cryptography.
- The connection is reliable. Message transport includes a message integrity check using a keyed Message Authentication Code (MAC). Secure hash functions are used for MAC computations.
  [Ref. 12]

SSL provides data encryption as well as authentication. It uses asymmetric cryptography, through RSA, Diffie-Hellman or Fortezza's Key Exchange Algorithm (KEA), to exchange a symmetric key. The symmetric key is then used to encrypt data for transmission. SSL includes the following cryptographic functionality:

- Encryption
  Symmetric: DES, RC4
  Asymmetric: RSA, DSS

- Authentication
  Certificates: X.509
  [Ref. 12]

It is the responsibility of the SSL Handshake protocol to coordinate the states of the client and server. An SSL session consists of a session identifier, a peer certificate, a compression method and a cipher specification. The SSL session may include multiple secure connections, i.e., each connection using the same session identifier. This is accomplished by allowing sessions to be resumable, as indicated by the session identifier and the **is_resumable** flag as described below. Additionally, clients and servers may be involved in multiple simultaneous sessions.

1. Record Layer

As mentioned above, the Record layer is the lowest of the SSL layers. The SSL Record Layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size. The record layer is responsible for fragmenting this data into *SSLPlaintext* records of 214 bytes or less. These records are composed of unencrypted text, which will be compressed and encrypted later. Client message boundaries are not preserved in the record layer. Also, application data is generally of lower precedence for transmission than other content types.

The Record layer is also responsible for compression and decompression of data. All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm. Initially it is defined as *CompressionMethod.null*, indicating that there is no compression. A *CompressionMethod.null* operation is an identity operation, no fields are actually altered. This operation is simply a means of insuring data is in the proper format (*SSLCompressed*) without actually doing compression. The compression algorithm translates an *SSLPlaintext* structure into an *SSLCompressed* structure, indicating that the structure is now compressed, even if the compression method was set to null. Compression must be lossless and may not increase the content length by more than 1024 bytes.

Decompression functions perform the opposite function on the receiving end. If the decompression function encounters a situation where the compressed structure would decompress to a length in excess of 214 bytes, it issues a fatal alert. Similarly, decompression functions are responsible for ensuring that messages cannot cause internal buffer overflows.

All records are protected using the encryption and MAC algorithms defined in the current **CipherSpec**. There is always an active **CipherSpec**, however, it is initially null, indicating that SSL security services are not being used. The handshake will decide upon an agreeable **CipherSpec** in order to gain security.

Once the handshake is complete, the two parties have shared secrets which are used to encrypt records and compute keyed MACs on their contents. The techniques used to perform the encryption and MAC operations are defined by the **CipherSpec**. The encryption and MAC functions translate an *SSLCompressed* structure into an *SSLCiphertext*, indicating that the structure is encrypted and secure. Transmissions also include a sequence number so that missing, altered, or extra messages are detectable.

a. Change CipherSpec Protocol

As mentioned above, the initial cipher suite for SSL is null, indicating that none of the desired MAC or encryption is taking place. The purpose of the Handshake Protocol is to negotiate a set of ciphers for a client and server. Thus, a means of changing from one **CipherSpec** to another is need. This is the purpose of the Change Cipher Spec protocol. The change cipher spec protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and compressed under the current (not the pending) **CipherSpec**.

The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated **CipherSpec** and keys. Reception of this message causes the receiver to copy the *read pending* state into the *read current* state. Separate read and write states are maintained by both the SSL client and server. When the client or server receives a *change cipher spec* message, it copies the pending read state into the current read state. When the client or server writes a *change cipher spec* message, it copies the *pending write* state into the *current write* state. The client sends a *change cipher spec* message following the handshake key exchange and certificate verify messages (if any), and the server sends a *change cipher spec* message after successfully processing the key exchange message it received from the client. An unexpected *change cipher spec* message should generate an alert.

2. Handshake Protocol

The cryptographic parameters of the session state between the client and server are negotiated by the SSL Handshake Protocol, which operates on top of the SSL Record Layer. When a SSL client and server first start communicating, they agree on parameters such as protocol version, cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. These processes are performed in the handshake protocol, which is summarized below.

The client sends a *client hello* message to which the server must respond with a *server hello* message, or else a fatal error will occur and the connection will fail. The **CipherSuite** list, passed from the client to the server in the *client hello* message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (first choice first). The server will select a cipher suite or, if no acceptable choices are presented, return a handshake failure alert and close the connection.

The *client hello* also includes a list of compression algorithms supported by the client, ordered according to the client's preference. If the server supports none of those specified by the client, the session must fail.

The *client hello* and *server hello* are used to establish the **CipherSpec**. The *client hello* and *server hello* establish the protocol version, session ID, cipher suite, and compression method. Two random values are also generated and exchanged which are later used to generate keys.

Following the *hello* messages, the server will send its certificate, if the server is to be authenticated. If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. These certificates are signed by a Certificate Authority that both the client and server recognize.

Now the server will send the *server hello done* message, indicating that the *hello*-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request message, the client must send either the certificate message or an alert stating that it has no certificate.

The *client key exchange* message is now sent, and the content of that message will depend on the public key algorithm selected between the *client hello* and the *server hello.* The choice of messages depends on which public key algorithm has been selected. If RSA, Diffie-Hellman or Fortezza are being used, a pre-master secret is generated and included in the *client key exchange* message. This pre-master secret is used for key generation as discussed below.

If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it under the public key from the server's certificate or temporary RSA key from a server key exchange message, and sends the result in an *encrypted premaster secret* message.

Under Fortezza DMS, the client derives a Token Encryption Key (TEK) using Fortezza's Key Exchange Algorithm (KEA). The client's KEA calculation uses the public key in the server's certificate along with private parameters in the client's token. The client sends public parameters needed for the server to generate the TEK, using its own private parameters. The client generates session keys, wraps them using the TEK, and sends the results to the server. The client generates Initializing Vectors (IVs) for the session keys and TEK and sends them also. The IV for the TEK is protected using the

KEA. The IV for the session key is protected by the TEK. The client also generates a random 48-byte premaster secret, encrypts it using the TEK, and sends the result to the server. This premaster secret is used to generate keys.

At this point, a *change cipher spec* message is sent by the client, and the client copies the pending **CipherSpec** into the current **CipherSpec**. The client then immediately sends the *finished* message under the new algorithms, keys, and secrets. In response, the server will send its own *change cipher spec* message, transfer the *pending* to the *current Cipher Spec*, and send its *Finished* message under the new *Cipher Spec*. At this point, the handshake is complete and the client and server may begin to exchange application layer data.

When mutually agreed upon, a SSL session may be resumed or duplicated. This is indicated by using a previously or currently active Session ID. The main reason for doing this is to avoid having to negotiate new security parameters. A session will end once all connections that are part of the session have terminated. The decision regarding how long a Session ID should be good for or whether it should be cached is up to system designers. When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows:

The client sends a *client hello* using the Session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a *server hello* with the same Session ID value. At this point, both client and server must send *change cipher spec* messages and proceed directly to finished messages. Once the re-establishment is complete, the client and server may begin to exchange application layer data. If a Session ID match is not found, the server generates a new session ID and the SSL client and server perform a full handshake.

3. Applying SSL to JMCIS'98 and Secure Password Transmission

SSL is a transport layer protocol which allows for the secure connections that are needed to transmit sensitive data and can be used in JMCIS'98 as an alternative to the Secure Password Transmission Protocol described earlier in this chapter.

SSL uses a handshake between the client and server to negotiate a cipher suite which includes the choice of asymmetric cryptography, symmetric cryptography and

Message Authentication Codes algorithms. SSL can easily be implemented in JMCIS'98 as it is a mature technology, supported and updated by Netscape and is in wide use on the Internet, especially to secure financial transactions.

In a JMCIS'98 context, SSL allows a client to connect to a gateway server, negotiate a set of cryptographic functions to secure the connection and then safely transmit sensitive information. SSL encrypts all data that flows across the connection. Thus all information will be secure. SSL allows for a client to authenticate a server before conducting secure transactions. Thus, the client can be comfortable with the identity of the server with which it is conducting transactions. SSL also allows for client authentication to the server. Therefore, SSL can be used to mutually authenticate the server and the client.

SSL does require a lot of overhead. For authentication a Certificate Authority and (at least) server-side certificates are required. We also mentioned above that optional client authentication was available. However, this would require each client to maintain an individual certificate. Additionally, the server would be required to verify the certificate which would become quite taxing on the system if usage were high. Caching Session IDs can help alleviate the problem, but is not a solution. During periods of high usage, mutual authentication would likely degrade system performance to the point that it would be unacceptable to a command and control system. If SSL were implemented we recommend only using server authentication. Note: client authentication has not been listed as being required nor desired.

F. SUMMARY

This chapter has introduced the Trusted Code and the Secure Password Transmission Protocols developed by Dennis Volpano. We then discussed how these protocols could be implemented using features of Java's Java Developer's Kit 1.1, the Java Cryptography Architecture and the Java Cryptography Extension. An alternative to the Secure Password Transmission protocol using Netscape's Secure Sockets Layer technology was also introduced and discussed. The following chapter gives our conclusions and recommendations for further research.

VI. CONCLUSIONS AND RECOMMENDATIONS

The goal of this thesis was to develop protocols to allow a client in a web environment to download and execute trusted code and to allow the client to securely transmit sensitive data such as a password to a gateway server.

Java provides mechanisms in the Java Developer's Kit 1.1, the Java Cryptography Architecture and the Java Cryptography Extension to implement these protocols. These features of Java are in their infancy and will surely change with time. However, Java has provided the APIs for the most common cryptographic mechanisms, including symmetric and asymmetric key cryptography, as well as Java implementations of some of them.

The functionality that is resident in these Java mechanisms allows for the type of cryptographic protection that are required by the Trusted Code and Secure Password Transmission protocols developed by Professor Volpano. These two protocols can be implemented using the features found in the Java Cryptography Architecture and the Java Cryptography Extension.

Chapter V introduced the Trusted Code and the Secure Password Transmission protocols which seek to securely carry out steps one through four of the JMCIS'98 Network Architecture (refer to Figure 1-1). These protocols solve the problem of clients downloading and executing trusted code from a server and secure password transmission from a client to a gateway server. Additionally, alternatives to the Trusted Code protocol using JDK 1.1 signed JAR files and to the Secure Password Transmission protocol using Netscape's Secure Sockets Layer were introduced and discussed.

Although Java is constantly evolving, especially in terms of security features and functionality, the mechanisms to support our protocols are readily available. We have made some assumptions in considering an operational implementation, namely that commercial web browsers would catch up to the JDK implementation discussed here.

The Trusted Code protocol can be implemented using the functionality provided in the Java Cryptography Architecture, specifically by using certificates and digitally signing the code that will be transmitted across the network. This approach can digitally sign any file that can then be verified as trustworthy by a client. The JDK 1.1 JAR file alternative involves packaging the files that will be transferred into a JAR file which is then digitally signed. Once a client receives this JAR file, the client will check to see whether the signer is listed in the client's ***javakey*** database as being trusted. If the signer is trusted, the code is accepted and can be executed. The JCA implementation can sign any type of file, whereas the JAR file approach will only sign a JAR file. Furthermore, the ***javakey*** database is command line driven and creating and maintaining it is somewhat cumbersome.

The Secure Password Transmission protocol can be implemented using the functionality of the Java Cryptographic Extension. Again, we have assumed for the sake of this thesis that an implementation of RSA in the JCE is forthcoming. The JCE allows a client to use asymmetric key cryptography (i.e., RSA) to securely pass an object consisting of a user's name, password, query and a symmetric key (i.e., DES) to a gateway server. The gateway server will then be able to pass query results back to the client encrypted with the symmetric key that each of them now share. The SSL alternative to this protocol is similar in that asymmetric cryptography is used to exchange a symmetric key that will be used for bulk encryption. However, in SSL the asymmetric cryptography is used only in the process of negotiating, creating and exchanging a pair of symmetric (session) keys, one for each direction of transmission. Once this has been accomplished, the symmetric keys are used to encrypt all of the data that will flow between the client and the server. Thus the SSL approach will encrypt all of the data that is flowing between the client and the server for duration of the session. SSL also allows for the client to be authenticated by the server, but this would entail the client maintaining a client-side certificate. We have assumed that the burden of creating and maintaining client certificates would be prohibitive in an environment such as JMCIS'98.

As stated in Chapter V, the biggest assumption made in this thesis is that commercial browser technology would catch up to the levels of JDK 1.1, the JCA and the JCE. This is not an unreasonable assumption. However, to be useful in a JMCIS'98 environment the browser(s) must implement these mechanisms as they are defined by Sun in the JDK. We have not assumed that all JMCIS'98 clients would use a single browser, however this is a logical assumption and would probably be best from a systems administrator point-of-view. Certainly, if these recommendations are to be implemented in JMCIS'98 there would need to be further research to ascertain how and when the functionality of the JDK 1.1, the JCA and the JCE 1.1 would be available in the commercial browsers.

There are also other areas that require further research and consideration before these protocols could be implemented on a large-scale. The choice of a certificate authority or certificate authorities is of concern. Research must be done to determine the type of certificates that would best suit the JMCIS'98 architecture and how those certificates should be generated and distributed.

Key management is also a critical issue. Depending on exactly which approach is taken with regard to the protocols presented in this thesis, key management will be a large part of the questions that must be answered in order to implement the protocols in a Navy-wide system. A key management infrastructure must be planned which will take into account such issues as how servers generate, distribute and revoke keys and certificates.

A determination should be made about how much security should be applied to areas such as trusted code and confidentiality within a system that is running in a system-high environment, as JMCIS'98 will be running over SIPRNET (Secret-level) and JWICS (Top Secret SCI-level). Adding the type of cryptographic functionality as described in this thesis will add to the overhead expense (i.e., administrative burden, time required for cryptographic functions, etc.,) of operating the system. A determination should be made at the highest levels of the JMCIS'98 arena that the system speed and performance degradation that would be involved in implementing any of these protocols, or their alternatives, can be reasonably absorbed in a command and control system.

It is also important to keep in mind that the protocols rely critically on being correctly implemented. We assume that the implementations of the Java API are correct. Any weakness in the implementation of the Java API results in a weakness in the protocols as well.

LIST OF REFERENCES

1. United States Navy Space and Naval Warfare Systems Command, *Joint Maritime Command Information System '98 (JMCIS'98) Single Acquisition Management Plan (SAMP)*, 1996, pp. 1-1 - 2-9.

2. United States Navy Space and Naval Warfare Systems Commad, *Joint Maritime Command Information System '98 (JMCIS'98) Security Policy and Requirements*, v. 0.1, 1997.

3. McGraw, G. and Felten, E., *Java Security : Hostile Applets, Holes and Antidotes*, John Wiley and Sons, New York, 1996.

4. Fritzinger, J. and Meuller, M., *Java Security*, Sun Microsystems, Inc., 1996.

5. Flanagan, D., *Java In A Nutshell, Second Edition,* O'Reilly & Associates, Sebastopol, California, 1997.

6. Sun Microsystems, Inc. *Java Security API Overview*, Sun Microsystems, Inc. 1997.

7. Sun Microsystems, Inc. *Java Cryptography Architecture API Specification and Reference*, Sun Microsystems, Inc., 1997.

8. Volpano, D. *"The Java Cryptography Architecture,"* CS3973 Class Notes, The Naval Postgraduate School, Monterey, California, 1997.

9. Sun Microsystems, Inc., *Java Cryptography Extension API Specification and Reference,* Sun Microsystems, Inc., 1997.

10. Volpano, D., *"The Java Cryptography Extension,"* CS3973 Class Notes, The Naval Postgraduate School, Monterey, California, 1997.

11. Volpano, D., Personal Communications, The Naval Postgraduate School, Monterey, California, 1996-1997.

12. Freier, A., Karlton, P. and Kocher, P., *The SSL Protocol, Version 3.0*," Netscape Communications, 1996.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Rd., Ste 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3. Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

4. Professor Dennis Volpano, Code CS/VO 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

5. Professor Cynthia Irvine, Code CS/IC 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

6. Michael E. Kono, Code 4221 1
Naval Command, Control and Ocean Surveillance Center RDT&E
53140 Gatchell Road
San Diego, CA 92152-5001

7. Commander 2
(Attn: Code N6)
Naval Security Group Command
9800 Savage Road
Fort George G. Meade, MD 20755-6000

8. Commanding Officer 2
(Attn: Code 30, CDR Zellman)
Naval Information Warfare Activity
9800 Savage Road
Fort George G. Meade, MD 20755-6000

9. Superintendent 1
(Attn: Code EC/LT)
Naval Postgraduate School
Monterey, CA 93943-5121

10. LT Steven G. Weldon 2
(Attn: Code N6)
9800 Savage Road
Fort George G. Meade, MD 20755-6000